A perspective on safety and real-time issues for GPU accelerated ADAS

Ignacio Sañudo Olmedo, Nicola Capodieci, Roberto Cavicchioli
Department of Physics, Informatics and Mathematics
University of Modena And Reggio Emilia
Modena, Italy
{name.surname}@unimore.it

***Abstract*—The current trend in designing Advanced Driving Assistance System (ADAS) is to enhance their computing power by using modern multi/many core accelerators. For many critical applications such as pedestrian detection, line following, and path planning the Graphic Processing Unit (GPU) is the most popular choice for obtaining orders of magnitude increases in performance at modest power consumption. This is made possible by exploiting the general purpose nature of today's GPUs, as such devices are known to express unprecedented performance per watt on generic embarrassingly parallel workloads (as opposed of just graphical rendering, as GPUs where only designed to sustain in previous generations). In this work, we explore novel challenges that system engineers have to face in terms of real-time constraints and functional safety when the GPU is the chosen accelerator. More specifically, we investigate how much of the adopted safety standards currently applied for traditional platforms can be translated to a GPU accelerated platform used in critical scenarios.**

I. Introduction

The recent trend in the automotive industry is the radical shift in how new vehicles are designed. New passenger vehicles require a technological transition to satisfy the computational demand of new-generation automotive software; this leads to novel research opportunities. All the big players in the automotive domain are spending a considerable amount of resources in ADAS (Advance Driver-Assistance System) development. Major OEMs (Original Equipment Manufacturer) like BMW, Volvo, Tesla, or General Motors and Tier-1s such as Bosch or Continental, are already building the necessary know-how and technological background to design the next generation of autonomous vehicles. This technological trend leads towards the integration of multiple applications with different criticality levels onto the same computing platform, thus considerably reducing production costs and power consumption requirements. This is accomplished by exploiting modern multi/many-core accelerators for the integration of applications featuring mixed-criticality real-time constraints. The intrinsic complexity of such architectures and the increasing complexity of the computing algorithms they have to sustain, poses significant and unprecedented challenges for both the real-time community and for functional safety engineers. As far as safety is concerned, standards have been published to guide the system engineers to overcome such challenges. In the automotive domain the development of new generation hardware and software components is growing very fast, however, it is unclear whether OEMs and Tier-1 players are creating prototypes meeting the constraints required by the standards, potentially creating unsafe products. Probably the best-known case so far is the Toyota unintended acceleration problem [1]. Traditionally, many of the mechanisms and safety measures used to mitigate potential risks have been developed and certified for single-core processors. Consequently, the designer of critical applications for multi/many-core systems has to be aware of unprecedented issues such as shared resource management and fault propagation. Moreover, while the real-time literature on single processing core is abundant, important issues such as response time analysis and achieving predictability when using modern compute accelerators represents a novel field of research that is getting more and more attention from system engineers and integrators. In this context, we decided to focus on the GPU, as it represents a very popular choice for achieving a high performance per watt ratio for complex embarrassingly parallel workloads that are typical of ADAS applications. The novelty of this presented paper is twofold: we first provide an exhaustive literature review on the current pitfalls for predictability and threats to real-time guarantees of GPU accelerated platforms. Then we discuss how the currently adopted traditional approaches to functional safety might be transposed in GPU accelerated safety critical platforms. These are issues that are impossible to ignore, as traditional safety-critical platforms cannot sustain the required compute power needed for safely function as expected, hence the need of accelerators that are most commonly not designed with real-time and safety issues as primary design goals. This paper is organized as follows:
In section II we provide a brief description on both hardware and software perspective on how modern GPUs are implemented. Section III summarizes the current efforts on issues related to both real-time and functional safety and in the subsequent Section IV, we provide insights on current and plausible future efforts to be done in order to implement such effort on GPU accelerated platforms. Section V concludes the paper with final remarks.

[1] NHTSA Report on Toyota Unintended Acceleration Investigation available at https://one.nhtsa.gov/About-NHTSA/Press-Releases/ci.NHTSA%E2%80%93NASA-Study-of-Unintended-Acceleration-in-Toyota-Vehicles.print%7D%7D

II. The Graphic Processing Unit

In this section, we describe the most common architectural followed for GPUs software and hardware design. This allows us to identify the areas in which real-time and functional safety aspects are involved.

A. Hardware Perspective

From the point of view of the hardware, a GPU is a hybrid between a SIMT (Single Instruction Multiple Threads) and SIMD (Single Instruction Multiple Data) heterogeneous chip multi-processor. Even if at the time of writing multiple GPU vendors are present, a high-level abstraction on the most fundamental function blocks of a GPU can be summarized as seen in Figure 1.

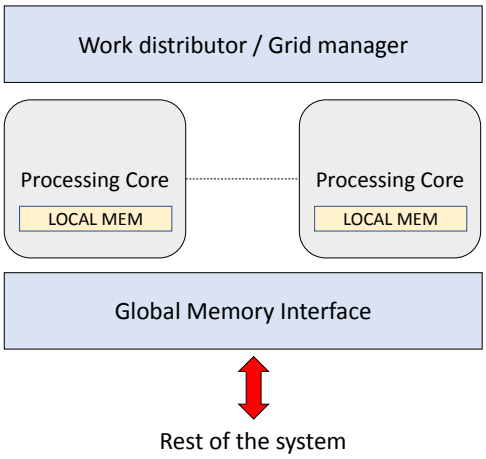

Figure 1. A high level block diagram for a generic GPU hardware architecture

In Figure 1 a GPU is shown as a collection of processing cores. Work offloaded to the GPU is therefore distributed among processing cores in order to exploit parallelism and, at the same time, to leverage efficient work balancing strategies. Within each processing core, lower level schedulers dispatch groups of threads in a *single instruction multiple data* fashion. Threads belonging to the same core can rely on local, low-latencies caches (for both instruction and data) as well as registers and scratchpads. These latter ones are often used for communication and synchronization procedures involving threads operating within the same processing core. Processing cores are able to communicate with each other via a global memory interface, which is also the interconnection fabric that connects the GPU to the rest of the system. Trivially, different vendors provide different implementations of all the blocks in Figure 1: more specifically, cache hierarchy, implementation details of the HW scheduling mechanisms, instruction decoding, branch prediction and ALU (Arithmetic Logic Units) pipelines might present significant differences. With respect to terminology, the processing cores in NVIDIA architectures are named *Streaming Multiprocessors (SMs)*[2], whereas for recent ARM[3] designed integrated GPUs, as well as Intel integrated graphics devices, the term *Execution Unit (EU)* is used[1]; AMD uses the term *(Next-Generation) Compute Unit (CU/NCU)*[4]. Depending on how a GPU is connected to the rest of the system, a GPU can be implemented as an integrated device with respect to the CPU (iGPU) or as a discrete peripheral (dGPU). iGPUs are commonly found in embedded SoCs (System on Chips) and in these solutions, global memory is typically shared with the CPU complex. Conversely, dGPUs are separate devices, connected through PCI-ex links; dGPUs, therefore, feature their own memory banks (VRAM) physically separated from the system RAM. The standard type of VRAM for discrete devices are usually GDDR RAM or HBM, as opposed to LPDDR for iGPUs.

B. Software Perspective

On a Software perspective, managing a GPU application forces the programmer to deal with a heterogeneous system, in which data is usually generated CPU-side and offloaded alongside a so called *description of work* to the GPU for its execution. As an aid to the programmer, APIs (Application Programming Interfaces) have been proposed by both device vendors and consortia (e.g. The Khronos Group): well known APIs for GPU programming are OpenGL, mostly for graphical rendering workloads, and CUDA or OpenCL for compute workloads. For the latter, the *description of work* to be executed on the GPU is called *kernel*, and data buffers visible within the GPU space are called *device buffers*. A GPU application, therefore, is usually composed by a buffer allocation management phase, data movements from the CPU-side to the GPU-side (and vice-versa) and kernels' invocations. The programmer influences how the GPU HW Work distributor acts by specifying kernel launch configurations: APIs like CUDA and OpenCL allow the user to define a compute grid in which parallel threads are logically grouped into blocks. Therefore, given a specific time instant, a single block of threads is in execution on one processing core.

III. Real-Time issues for GPU accelerators

Designing a Real-Time system means designing a predictable platform. In other words, the combination of both software and hardware components must be designed so to have critical applications running with known and bounded timing requirements. These aspects are evaluated using timing analysis, i.e. each critical task is modeled as a periodic workload, characterized by a Worst Case Execution Time (WCET), a deadline and a period. Generally, the specification of the timing requirements at the software level is derived using exact schedulability analysis. Studying the Real-Time behavior of a modern GPU, therefore, implies understanding what are the possible threats for estimating WCETs and how to control GPU application scheduling.

[2]NVIDIA GV100 White Paper http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

[3]ARM MALI G72https://developer.arm.com/products/graphics-and-multimedia/mali-gpus/mali-g72-gpu

[4]AMD VEGA White paper https://radeon.com/_downloads/vega-whitepaper-11.6.17.pdf

A. WCET Estimation

Threats to WCET estimation on GPU-accelerated platforms are represented by three factors: (1) estimation of time to completion for the execution paths and branches in GPU kernels, (2) CPU interactions with the driver stack and (3) memory contention.

As far as (1) is concerned, estimating the WCET out of a program to be executed on a traditional uniprocessor system is a well-understood problem [2]. However, traditional techniques for static and dynamic code analyses do not have a straightforward application to the GPU due to the SIMD-lockstep scheduling mechanisms within GPU threads. More specifically, the minimal schedulable entity within a GPU processing core is a plurality of threads (usually a multiple of 16), grouped in a *warp* or *wavefront* in CUDA and OpenCL nomenclature respectively. Instructions are therefore fed to a warp/wavefront and execution within parallel threads happens in lockstep. In this scenario, branch divergence inside a warp/wavefront has a significant performance hit and there is no direct correspondence of this in a multicore CPU. In literature such issues have been addressed with novel methodologies such as hybrid analysis [3] and extreme value theory characterization [4]. Compute kernels are meant to solve problems that are intrinsically embarrassingly parallel, hence significant branch divergence or multiple nested conditions are difficult to find in a program to be executed on a GPU. Due to the dramatic impact on performance [5], the system engineer that faces the implementation of highly irregular kernels would likely opt for an alternative CPU execution.

CPU interactions (2) with the GPU subsystem play a significant role in predictability. Most common APIs such as CUDA and OpenCL rely on a constant interaction between a CPU process and the GPU. CPU applications communicate with the device driver in order to translate API function calls into GPU commands. These commands are then streamed to the GPU processing cores. GPU commands are usually classified into synchronous and asynchronous with respect to the CPU host. Synchronous commands imply having a CPU thread waiting until the completion of such commands before operating other tasks. This wait can be an active spinning on a shared synchronization construct or managed via interrupt requests (IRQs). Active polling on a shared resource is considered wasteful for both CPU utilization and power consumption, although it minimizes latencies, whereas IRQs management allows the CPU core to perform other work but this approach introduces further scheduling issues. For instance, Elliot et al. in [6] realized that standard Linux interrupt handling involves heuristics that can potentially lead to long latencies and even priority inversion phenomena. In order to mitigate this, Elliot et al. presented a flexible real-time interrupt handling techniques for multiprocessor platforms that are applicable to any JLSP-scheduler (Job-Level Static-Priority), hence allowing for implementing scheduling mechanisms for IRQ routines able to support both fixed-priority and deadline based approaches. The dramatic variability in WCETs introduced by CPU driver interactions was also noticed in other research work, most notably in [7] and [8].

Memory interference (3) represents the most concerning aspect in achieving predictability in GPU-accelerated SoCs. Memory contention affects the most common embedded designs in which the GPU and the multicore host shares system RAM (and sometimes even Last Level Caches). When both these clients access memory, contention at the level of the memory controller and within DRAM banks can cause a significant performance degradation to the observed applications, as concurrent access to memory devices is most commonly arbitrated with no real-time compliant mechanisms. In [9] a complete evaluation of the extent of memory contention is presented: Cavicchioli et al. show that a GPU application can experience performance degradation up to 100% in case of intensive memory use from a CPU application and, specularly, performance degradation of a CPU application with an interfering GPU memory bounded activity can cause latencies increase of almost 6x. Tests were conducted in multiple commercial SoCs that features an integrated GPU, such as NVIDIA development boards (TX1 and TK1) and an Intel i7 processor. Several approaches have been proposed to deal with memory interference in heterogeneous platforms. These solutions are most often designed for providing means for memory centric scheduling, i.e. tasks are assumed to be compliant with a predictable execution model (PREM) [10] that separates memory phases from computation phases. In such an execution model, the memory phase involve accessing shared memory for loading/unloading data into caches and/or scratch pad memories that are local and for exclusive use of a CPU core. Compute phases, are then allowed to compute on those previously fetched data. By keeping a separation between memory and compute phase of a task, ad-hoc scheduling algorithms for shared memory concurrency can be applied to different memory phases of different applications. $Si\Gamma$ [11] is an example of a server based arbitration mechanism between CPU and GPU able to intercept CUDA function calls and CPU requests for memory phases and privileges CPU memory bounded applications. More recently, Ali and Yun in [12] presented a complementary mechanism that is able to protect GPU applications in case of CPU memory bounded interfering processes. In this latter contribution, CUDA function calls are intercepted and a CPU memory access throttling mechanism based on MEMGUARD [13] is applied. While it is trivial to understand how a CPU application can be coded in a PREM-compliant fashion, challenges arise when coding PREM compliant GPU kernels. Forsberg et al. in [8] address this problem with CUDA warp specialization: the traditional way to write a CUDA kernel is to have both memory and compute instructions mixed within the same warp, whereas in the warp specialization approach, we define memory and compute warps. A memory warp fetches memory from the shared DRAM and stores it into the local scratch pad memory within each SM; trivially, a memory warp can also flush data stored in the local scratch pads to central memory. Compute warps are mostly allowed to compute on

data previously fetched when memory warps were scheduled. At the boundary of each memory warp, a scheduling entity (usually implemented at hypervisor or operating system level) is contacted and thus software throttling mechanisms (for both the CPU and GPU) are applied. In all the cited proposed solutions, software mechanisms are put in place in order to apply the desired level of memory access throttling. In contrast and in order to solve the same problem of arbitrating memory accesses triggered by both CPU and GPU, Houdek et al. [14] exploit hardware throttling for memory clients that acts at the level of memory controller.

B. Real Time GPU Scheduling

At the time of writing, the problem with GPU scheduling is that GPU hardware vendors tend to favor closed source driver implementations and avoid providing details on how a system designer can interact on scheduling aspects. To face this problem alternative open source GPU driver implementation have been exploited to provide a predictable scheduling policy on GPUs. Kato et al., for instance, presented TimeGraph [15], a non-preemptive fixed-priority scheduler for graphic GPU tasks. Schnitzer et al. [16] proposed a Reservation-based scheduling mechanism able to schedule graphics tasks of an automotive application to meet frame-rate constraints, such as the law mandatory speedometer rendering when only a virtual cockpit is available in a vehicle. As far as compute workloads are concerned, a fruitful line of research in GPU scheduling is represented by Persistent Threads programming paradigm [17], [18], [19], [20]. This model allows the implement the scheduling policy of choice by batching many kernel calls into a single invocation to then apply arbitrary scheduling decisions to blocks of GPU threads within one or more persistently executing GPU threads. Synchronization among scheduled blocks of threads is made possible with the implementation of fast barriers [21]. No matter which scheduling approach we elect to use, the biggest obstacle in implementing efficient scheduling algorithms is given by the assumption of absent or limited preemption capabilities for the GPU engines. Recently, starting from the NVIDIA Pascal GPU architecture[5], instruction level preemption granularity is possible and this will most likely lead to more efficient and predictable scheduling algorithms for the GPU. As of now, in all the previously cited contributions with respect to GPU scheduling, preemptive scheduling is applied with a preemption granularity that corresponds to a block of threads, i.e. CTA level preemption (Cooperative Thread Array, in CUDA) and workgroups (in OpenCL terminology).

IV. Functional Safety for GPUs

While for discussing real-time issues we can only rely on published research experience, functional safety is a topic that is regulated and described by well known and accepted regulation standards. According to IEC61508 [22], functional safety is defined as follows: "Freedom from unacceptable risk of physical injury or of damage to the health of people, either directly, or indirectly as a result of damage to property or to the environment". Many software standards in automotive define guidelines to evaluate and support systems executing applications with different criticality levels. Although these standards describe how to develop applications complying with software safety assessments, some of the safety recommendations in terms of software implementation are left to the interpretation of the system engineer. A clear example of this is the requirement to achieve complete spatial and temporal isolation of resources within partitions of a mixed-criticality system: standards demand to achieve this, without specifying the implementation details of such effort.

ISO-26262 is the main functional safety standard for the development of electronic systems in passenger vehicles. The standard works as a guidance to avoid risks due to hazards caused by malfunctioning behavior of the vehicle. This standard defines the entire life-cycle of a functional safety automotive component, spanning from management and development to production, decommissioning and relation with suppliers. Given the space constraint of our contribution, in this section we will not cover the entire production cycle and design methodologies; instead, we will focus on runtime mechanisms put in place in order to guarantee safety on a GPU accelerated platform. More specifically, among all the requirements recommended by ISO-26262, we will put our attention on GPU-side hardware and software approaches to achieve redundancy and Freedom from Interference (FFI).

A. FFI on GPUs

The integration of multiple applications in the same platform calls for the need to implement safety-related and non-safety-related applications in the same multicore System on Chip (MPSoC). If the analyzed platform features a GPU as compute accelerator, the problem of managing applications belonging to different criticality level calls for further discussion.

In order to comply with technical safety requirements, *Freedom from Interference (FFI)* plays a key role. According to the annex D of ISO-26262, "FFI is the absence of cascading failures between two or more elements that could lead to the violation of a safety requirement". In this context, software components cannot be physically separated, as all the applications execute in the same MPSoC, therefore FFI represents a design criticality. Generally, achieving FFI between partitions allows deploying software components with different ASIL (Automotive Safety Integrity Level) grades. In this way, lowest ASIL components involved in the certification process do not inherit the safety level of highest ASIL software components (ASIL lift-up), reducing design and production costs. At implementation level, software components might belong to different criticality domains and if they are allowed to share resources (such as CPU, GPU time, but also memory devices) with no ad-hoc safety mechanisms in place, interference can threaten performance, predictability (as already pointed out in the previous section) and security. We will focus on the

[5]NVIDIA GP100 White Paper: https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf

requirements of spatial and temporal isolation as they are defined in the standard.

It is important to notice that virtualization technologies (e.g: type 1 hypervisors), are a perfect candidate for meeting the isolation requirements recommended by the ISO-26262: "Virtualization technologies can support the argument to guarantee freedom from interference between software elements running on a multi-core platform."

Temporal isolation mandates the necessity to ensure that one application does not interfere with another application in terms of execution time: for instance, consuming its CPU cycles or blocking a shared resource used by another application as this might lead to deadlocks or livelocks. According to the standard, the strategies that can be considered to avoid temporal interference are: cyclic execution scheduling, fixed priority based scheduling, time triggered scheduling, monitoring of processor execution time, program sequence monitoring and arrival rate monitoring. Analyzing a single core of a CPU, applications within the same Virtual Machine (VM) are arbitrated by means of an operating system scheduler, whereas among different VMs temporal isolation is achieved by pinning VMs to virtual CPUs binded to a single physical core. Virtual CPUs within a physical core are commonly arbitrated with scheduling algorithms dictated by the hypervisor. Hence, hierarchical scheduling is applied. Trivially, if VMs are pinned to different CPU cores, temporal isolation from different CPUs is granted, however interference arise in case of shared memory subsystem (i.e. shared caches and shared system RAM). Inter-core interference in CPU shared caches is a well known problem and it has been abundantly addressed before in literature. For instance in [23], [24], authors of such contributions proposed memory page coloring and cache lockdown mechanisms to enforce a deterministic cache hit rate on the most frequently accessed memory pages.

Temporal isolation for the GPU is strictly related to GPU scheduling and this topic has been treated in Section III-B. The issue to discuss is how virtualization solutions are applied to a GPU [25]. On this purpose, a fairly complete and recent survey on GPU virtualization can be found in [26]. According to [26], there are three different methodologies for enabling GPU virtualization: if the virtualization engineer has no access to the GPU driver stack, GPU API calls might be intercepted and rerouted to a virtualized privileged domain (or even a remote host), which in turn, can arbitrate GPU access requests coming from different VMs (API remoting). Para and Full-virtualization imply modifying the GPU driver installed in the VMs so to allow them to be able to access the GPU HW only for non-critical operations, whereas for allocations and scheduling commands a virtualized privileged domain act as an arbiter. Finally, specific HW extension might be implemented within the SoC/GPU so to provide hardware assisted virtualization. For small-footprint hypervisors, that are typical of the embedded world, para-virtualization is the most promising solution, whereas the other two approaches are now widely adopted in the High Performance Computing (HPC) world. A fairly known example of GPU para-virtualization is represented by XenGT [27]. Spatial isolation within a GPU becomes a topic to discuss as we realize the massively parallel nature of such an accelerator. As we highlighted in section II, a GPU is internally subdivided into different processing cores and a user might want to allocate or reserve a subset of those processing cores to different applications belonging to different criticality levels. This approach is convenient only for the HPC-oriented devices, as the number of their processing cores can scale to a very large number (e.g. an NVIDIA Tesla P100 features 60 SMs). For embedded platforms, however, as a completely different power consumption and die size is needed for automotive scenarios, the GPU is sufficiently small to be considered as a single computing resource, in which we consider one GPU task at a time, hence taking advantage of thread-level parallelism within the application, but not among different tasks. The performance benefits of allowing multiple applications to run in parallel in multiple GPU processing cores would be threatened by the isolation problems caused by GPU self-interference [28].However, the future possibility of having SoC featuring sufficiently large iGPUs to allow partitioning of GPU processing core calls for preliminary safety-related studies: in this case, GPU scheduling for providing temporal isolation will have to take inspiration from Multi-Processor scheduling and at the same time, interference on GPU shared caches will become a problem. Preliminary investigations (based on simulators) on applying cache reservation mechanisms on the GPU have been recently proposed in [29]. In this latter contribution, authors argue that GPU L2 cache locking improves predictability at a negligible performance hit. In the embedded automotive context, these findings represent a promising direction for implementing established methodologies for GPU resource partitioning that are now common in HPC-oriented scenarios. A documented example of this is the NVIDIA CUDA MultiProcess Service (MPS)[6], that allows for simultaneous execution of multiple CUDA contexts within the same GPU. In Figure 2 (a), we can see how GPU time might be contended between different applications in a scenario in which we allow the execution of one context at a time. In this case, there is a context switch between two applications, hence the last level-cache (LLC) is not sensible to inter-application interference. Time predictability, however, is sensible to the length of this context switch. In Figure 2 (b) shows the opposite situation: the GPU is divided into two partitions based on the number N of processing cores. In this scenario, parallel execution of contexts residing in different domains will compete for access in GPU LLC, threatening both performance and predictability when no cache locking or coloring mechanism is in place.

Critical memory regions are protected from unauthorized accesses with standard methodologies for memory protection (e.g. Memory Management Units) as they are implemented at SoC level and at GPU level [30], [31].

[6]CUDA MPS Documentation https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf

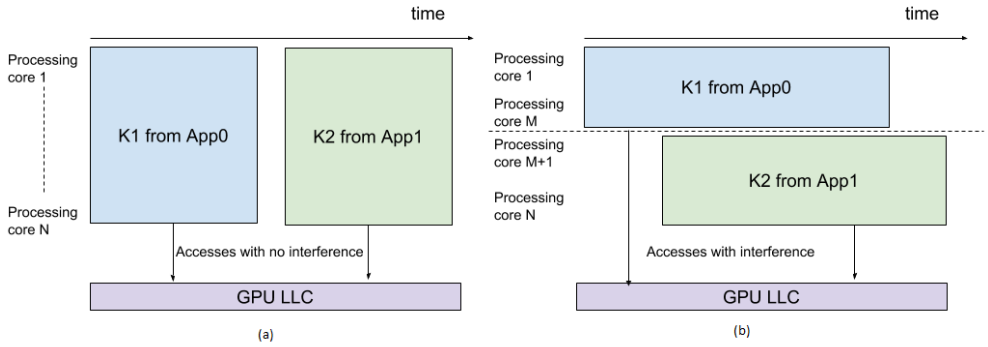

Figure 2. Timeline of temporal (a) and spatial (b) interference within a GPU. K1 and K2 are compute kernels invoked by applications App0 and App1 respectively.

B. Redundancy on GPUs

The need of redundancy on automotive grade platforms is how the ISO-26262 diversity requirement is most commonly interpreted. More specifically, ISO-26262 defines diversity as the system capability to provide "different solutions satisfying the same requirement with the aim of independence". In other words, computing devices should be able to sustain both random and systematic failures by using multiple solutions for the same problem. If we interpret the solution as the work of a computing device and the problem as a specific functionality (such as algorithms or even single instructions), it is then trivial to understand that substantial HW design efforts for computing devices must be put in place. According to [32], there are a number of possible design blueprints for safety-tolerant modern microprocessors. The easiest and cheapest solution is to obtain redundancy via software mechanisms: the same software artifact is compiled in different ways or different implementations of the same algorithms run in the same CPU core, then intermediate results might be compared with a Multiple Input Shift Register (MISR). If a higher ASIL level is desired, a plausible solution is represented by two identical CPUs in parallel lockstep, in which a master-slave configuration allows for instruction consistency checks. Other solutions imply having a co-processor to be sided with the master CPU that is not necessarily a replica of the master core, but instead acts a highly optimized supervisor. All these different solutions are aimed to obtain different degrees of safety guarantees (often measured in terms of ASIL grade) at a variable cost of production, implementation and overall performance degradation. For transposing these approaches to a GPU, previous research efforts were mostly aimed at providing redundancy within a GPU in order to improve reliability in HPC-oriented applications. From a software perspective, Dimitrov et al. in [33], exploit existing GPU APIs for providing application-level mechanisms for safety. Example of such approaches are duplicating kernel invocations in both sequential and concurrent fashion. On hardware side, in [34], Sheaffer et al., exploit the concept of Architectural Vulnerability Factor (AVF), which is a formula able to estimate the likelihood that a transient fault in a particular subsection of a GPU will result in a computational error. According to the AVF value, the functional safety engineer will select the appropriate granularity within a GPU to apply redundancy. Redundancy in GPUs, therefore, implies reserving the use of a portion of GPU computing resources to act in the same way as a replicated CPU or supervisor co-processors. Tests regarding fault tolerance for GPU automotive applications generally involves testing their robustness through fault injection [35]. Redundancy granularity can be set at the level of entire processing cores, warps/wavefronts and even within ALUs. Even if all these solutions have been evaluated in HPC scenarios, they still represent a promising direction for embedded ADAS reliable applications. Moreover, ECC/parity bits for memory protection (both local and global to the GPU processing cores) is nowadays a very common feature for both HPC and embedded oriented GPUs. The presence of such protection mechanisms is mandatory due to the average size of input data of automotive applications, which are also known to be strongly memory bounded.

V. Conclusion

In this work we presented and discussed how commonly adopted methodologies to achieve real-time guarantees and safety assurances can be transposed from traditional CPU systems to heterogeneous systems, such as GPU accelerated MPSoCs. As far as real-time issues are concerned, fruitful research directions on both GPU kernel scheduling and memory interference are getting more and more attention. We provided an extensive survey on such topics so that the most common approaches have been thoroughly described. We argue that methodologies for GPU scheduling and memory interference arbitration in SoCs will play a key role for the automotive industry players: these are not only important from a real-time perspective, but it is also a necessity dictated by the requirements of both temporal and spatial isolation as recommended by ISO-26262. We also highlighted how methodologies for error detection and redundancy for GPUs are still inspired by what has been presented in CPU-related literature: moreover, we argue that even if such mechanisms

are well established for HPC oriented devices, these very same approaches can also be extremely useful in embedded automotive scenarios [36].

ACKNOWLEDGMENT

This work is part of the Hercules and I-MECH (Intelligent Motion Control Platform for Smart Mechatronic Systems) projects, which are respectively funded by the EU Commission under the HORIZON 2020 framework program (GA-688860) and by ECSEL JA 2016 research and innovation program under grant agreement No. 737453.

REFERENCES

[1] Intel, "The compute architecture of intel processor graphics gen9, v. 1.0," *Intel White Paper*, 2015. [Online]. Available: https://software.intel.com/sites/default/files/managed/c5/9a/The-Compute-Architecture-of-Intel-Processor-Graphics-Gen9-v1d0.pdf

[2] P. Puschner and C. Koza, "Calculating the maximum execution time of real-time programs," *Real-time systems*, vol. 1, no. 2, pp. 159–176, 1989.

[3] A. Betts and A. Donaldson, "Estimating the wcet of gpu-accelerated applications using hybrid analysis," in *Real-Time Systems (ECRTS), 2013 25th Euromicro Conference on*. IEEE, 2013, pp. 193–202.

[4] K. Berezovskyi, L. Santinelli, K. Bletsas, and E. Tovar, "Wcet measurement-based and extreme value theory characterisation of cuda kernels," in *Proceedings of the 22nd International Conference on Real-Time Networks and Systems*. ACM, 2014, p. 279.

[5] J. Meng, D. Tarjan, and K. Skadron, "Dynamic warp subdivision for integrated branch and memory divergence tolerance," *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3, pp. 235–246, 2010.

[6] G. A. Elliott and J. H. Anderson, "Robust real-time multiprocessor interrupt handling motivated by gpus," in *Real-Time Systems (ECRTS), 2012 24th Euromicro Conference on*. IEEE, 2012, pp. 267–276.

[7] G. A. Elliott, "Real-time scheduling for gpus with applications in advanced automotive systems," Ph.D. dissertation, The University of North Carolina at Chapel Hill, 2015.

[8] B. Forsberg, L. Benini, and A. Marongiu, "Heprem: Enabling predictable gpu execution on heterogeneous soc," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2018*. IEEE, 2018, pp. 539–544.

[9] R. Cavicchioli, N. Capodieci, and M. Bertogna, "Memory interference characterization between cpu cores and integrated gpus in mixed-criticality platforms," in *22nd IEEE International Conference on Emerging Technologies And Factory Automation (ETFA)*, 2017.

[10] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley, "A predictable execution model for cots-based embedded systems," in *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE, 2011, pp. 269–279.

[11] N. Capodieci, R. Cavicchioli, P. Valente, and M. Bertogna, "Sigamma: server based integrated gpu arbitration mechanism for memory accesses," in *Proceedings of the 25th International Conference on Real-Time Networks and Systems*. ACM, 2017, pp. 48–57.

[12] W. Ali and H. Yun, "Protecting real-time gpu applications on integrated cpu-gpu soc platforms," *arXiv preprint arXiv:1712.08738*, 2017.

[13] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, "Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms," in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th*. IEEE, 2013, pp. 55–64.

[14] P. Houdek, M. Sojka, and Z. Hanzálek, "Towards predictable execution model on arm-based heterogeneous platforms," in *Industrial Electronics (ISIE), 2017 IEEE 26th International Symposium on*. IEEE, 2017, pp. 1297–1302.

[15] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa, "Timegraph: Gpu scheduling for real-time multi-tasking environments," 2011.

[16] S. Schnitzer, S. Gansel, F. Dỳrr, and K. Rothermel, "Real-time scheduling for 3d gpu rendering," in *Industrial Embedded Systems (SIES), 2016 11th IEEE Symposium on*. IEEE, 2016, pp. 1–10.

[17] J. Breitbart, "Static gpu threads and an improved scan algorithm," in *European Conference on Parallel Processing*. Springer, 2010, pp. 373–380.

[18] N. Capodieci and P. Burgio, "Efficient implementation of genetic algorithms on gp-gpu with scheduled persistent cuda threads," in *Parallel Architectures, Algorithms and Programming (PAAP), 2015 Seventh International Symposium on*. IEEE, 2015, pp. 6–12.

[19] G. Chen, Y. Zhao, X. Shen, and H. Zhou, "Effisha: A software framework for enabling effficient preemptive scheduling of gpu," in *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 2017, pp. 3–16.

[20] B. Wu, X. Liu, X. Zhou, and C. Jiang, "Flep: Enabling flexible and efficient preemption on gpus," in *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2017.

[21] S. Xiao and W.-c. Feng, "Inter-block gpu communication via fast barrier synchronization," in *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*. IEEE, 2010, pp. 1–12.

[22] "Functional safety of electrical/electronic/programmable electronic safety-related systems," International Organization for Standardization, Geneva, CH, Standard, 2000.

[23] M. Caccamo, M. Cesati, R. Pellizzoni, E. Betti, R. Dudko, and R. Mancuso, "Real-time Cache Management Framework for Multi-core Architectures," in *Proceedings of the 2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, ser. RTAS '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 45–54. [Online]. Available: http://dx.doi.org/10.1109/RTAS.2013.6531078

[24] R. Tabish, R. Mancuso, S. Wasly, A. Alhammad, S. S. Phatak, R. Pellizzoni, and M. Caccamo, "A Real-Time Scratchpad-Centric OS for Multi-Core Embedded Systems," in *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2016, pp. 1–11.

[25] Y. Suzuki, S. Kato, H. Yamada, and K. Kono, "Gpuvm: Why not virtualizing gpus at the hypervisor?" in *USENIX Annual Technical Conference*, 2014, pp. 109–120.

[26] C.-H. Hong, I. Spence, and D. S. Nikolopoulos, "Gpu virtualization and scheduling methods: A comprehensive survey," *ACM Computing Surveys (CSUR)*, vol. 50, no. 3, p. 35, 2017.

[27] H. Shan, K. Tian, E. Dong, and D. Cowperthwaite, "Xengt: A software based intel graphics virtualization solution," *Xen Project Developer Summit, Edinburgh, Scotland*, 2013.

[28] A. Jog, O. Kayiran, T. Kesten, A. Pattnaik, E. Bolotin, N. Chatterjee, S. W. Keckler, M. T. Kandemir, and C. R. Das, "Anatomy of gpu memory system for multi-application execution," in *Proceedings of the 2015 International Symposium on Memory Systems*. ACM, 2015.

[29] J. Picchi and W. Zhang, "Impact of l2 cache locking on gpu performance," in *SoutheastCon 2015*. IEEE, 2015, pp. 1–4.

[30] J. Power, M. D. Hill, and D. A. Wood, "Supporting x86-64 address translation for 100s of gpu lanes," in *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*. IEEE, 2014, pp. 568–578.

[31] T. Zheng, D. Nellans, A. Zulfiqar, M. Stephenson, and S. W. Keckler, "Towards high performance paged memory for gpus," in *High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on*. IEEE, 2016, pp. 345–357.

[32] M. Bellotti and R. Mariani, "How future automotive functional safety requirements will impact microprocessors design," *Microelectronics Reliability*, vol. 50, no. 9-11, pp. 1320–1326, 2010.

[33] M. Dimitrov, M. Mantor, and H. Zhou, "Understanding software approaches for gpgpu reliability," in *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*. ACM, 2009, pp. 94–104.

[34] J. W. Sheaffer, D. P. Luebke, and K. Skadron, "A hardware redundancy and recovery mechanism for reliable scientific computation on graphics processors," in *Graphics Hardware*, vol. 2007. Citeseer, 2007, pp. 55–64.

[35] R. Bramley, "Functional safety and the gpu," GPU Technology Conference, NVIDIA GTC 2017, 2017.

[36] P. Burgio, M. Bertogna, N. Capodieci, R. Cavicchioli, M. Sojka, P. Houdek, A. Marongiu, P. Gai, C. Scordino, and B. Morelli, "A software stack for next-generation automotive systems on many-core heterogeneous platforms," *Microprocessors and Microsystems*, vol. 52, pp. 299–311, 2017.